DataAllZeroes.cpp

#include "DataAllZeros.h"

/\*\*

\* Default constructor.

\*/

DataAllZeros::DataAllZeros() : DataGenerator("All zeroes") {}

/\*\*

\* Destructor.

\*/

DataAllZeros::~DataAllZeros() {}

/\*\*

\* Generate all zeros to fill a data vector.

\* @param data the data vector to fill.

\*/

void DataAllZeros::generate\_data(vector<Element>& data, int size)

{

for (int i = 0; i < size; i++) data.push\_back(Element(0));

}

DataAllZeros.h

#ifndef DATAALLZEROS\_H\_

#define DATAALLZEROS\_H\_

#include "DataGenerator.h"

#include "Element.h"

/\*\*

\* Generator for data that is all zeros.

\*/

class DataAllZeros: public DataGenerator

{

public:

DataAllZeros();

virtual ~DataAllZeros();

virtual void generate\_data(vector<Element>& data, int size);

};

#endif /\* DATAALLZEROS\_H\_ \*/

DataGenerator.cpp

#include <string>

#include <vector>

#include "DataGenerator.h"

using namespace std;

/\*\*

\* Constructor.

\* @param name the name of this generator.

\*/

DataGenerator::DataGenerator(string name) : name(name) {}

/\*\*

\* Destructor.

\*/

DataGenerator:: ~DataGenerator() {}

/\*\*

\* Getter

\* @return the name of this generator.

\*/

string DataGenerator::get\_name() { return name; }

DataGenerator.h

#ifndef DATAGENERATOR\_H\_

#define DATAGENERATOR\_H\_

#include <string>

#include <vector>

#include "Element.h"

using namespace std;

/\*\*

\* Abstract base class of the data generator subclasses.

\*/

class DataGenerator

{

public:

DataGenerator(string name);

virtual ~DataGenerator();

virtual string get\_name();

virtual void generate\_data(vector<Element>& data, int size) = 0;

private:

string name;

};

#endif /\* DATAGENERATOR\_H\_ \*/

DataRandom.cpp

#include <stdlib.h>

#include <time.h>

#include "DataRandom.h"

#include "Element.h"

/\*\*

\* Default constructor.

\*/

DataRandom::DataRandom() : DataGenerator("Unsorted random") {}

/\*\*

\* Destructor.

\*/

DataRandom::~DataRandom() {}

/\*\*

\* Generate random values to fill a data vector.

\* Reuse previously generated data if possible.

\* @param data the target data vector.

\* @param size the number of random data elements to generate.

\*/

void DataRandom::generate\_data(vector<Element>& data, int size)

{

data.clear();

// Reuse previously generated random data.

if (random\_data.size() == size)

{

data = random\_data;

}

// Generate new random data.

else

{

random\_data.clear();

srand(time(NULL)); // seed the random number generator

for (int i = 0; i < size; i++)

{

long r = rand()%size;

Element elmt(r);

random\_data.push\_back(elmt);

data.push\_back(elmt);

}

}

}

DataRandom.h

#ifndef DATARANDOM\_H\_

#define DATARANDOM\_H\_

#include "DataGenerator.h"

#include "Element.h"

/\*\*

\* Generator for unsorted random data.

\*/

class DataRandom: public DataGenerator

{

public:

DataRandom();

virtual ~DataRandom();

void generate\_data(vector<Element>& data, int size);

private:

vector<Element> random\_data;

};

#endif /\* DATARANDOM\_H\_ \*/

DataReverseSorted.cpp

#include "DataReverseSorted.h"

/\*\*

\* Default constructor.

\*/

DataReverseSorted::DataReverseSorted() : DataGenerator("Reverse sorted") {}

/\*\*

\* Destructor.

\*/

DataReverseSorted:: ~DataReverseSorted() {}

/\*\*

\* Generate reverse sorted values (high to low) to fill a data vector.

\* @param data the data vector to fill.

\*/

void DataReverseSorted::generate\_data(vector<Element>& data, int size)

{

for (long i = 0; i < size; i++) data.push\_back(Element(size - i));

}

DataReverseSorted.h

#ifndef DATAREVERSESORTED\_H\_

#define DATAREVERSESORTED\_H\_

#include "DataGenerator.h"

#include "Element.h"

/\*\*

\* Generator for reverse-sorted data.

\*/

class DataReverseSorted: public DataGenerator

{

public:

DataReverseSorted();

virtual ~DataReverseSorted();

virtual void generate\_data(vector<Element>& data, int size);

};

#endif /\* DATAREVERSESORTED\_H\_ \*/

DataSorted.cpp

#include "DataSorted.h"

/\*\*

\* Default constructor.

\*/

DataSorted::DataSorted() : DataGenerator("Already sorted") {}

/\*\*

\* Destructor.

\*/

DataSorted:: ~DataSorted() {}

/\*\*

\* Generate already sorted values (low to high) to fill a vector.

\* @param data the data vector to fill.

\*/

void DataSorted::generate\_data(vector<Element>& data, int size)

{

for (long i = 0; i < size; i++) data.push\_back(Element(i));

}

DataSorted.h

#ifndef DATASORTED\_H\_

#define DATASORTED\_H\_

#include "DataGenerator.h"

#include "Element.h"

/\*\*

\* Generator for already sorted data.

\*/

class DataSorted: public DataGenerator

{

public:

DataSorted();

virtual ~DataSorted();

virtual void generate\_data(vector<Element>& data, int size);

};

#endif /\* DATASORTED\_H\_ \*/

InsertionSort.h

#ifndef INSERTIONSORT\_H\_

#define INSERTIONSORT\_H\_

#include "VectorSorter.h"

/\*\*

\* The class that implements the insertion sort algorithm

\* for a vector of data.

\*/

class InsertionSort: public VectorSorter

{

public:

InsertionSort(string name);

virtual ~InsertionSort();

virtual void run\_sort\_algorithm() throw (string);

};

#endif /\* INSERTIONSORT\_H\_ \*/

LinkedList.h

#ifndef LINKEDLIST\_H\_

#define LINKEDLIST\_H\_

#include "Node.h"

/\*\*

\* The linked list class for the mergesort algorithm.

\*/

class LinkedList

{

public:

LinkedList();

LinkedList(Node \*head, Node \*tail, const int size);

virtual ~LinkedList();

Node \*get\_head() const;

Node \*get\_tail() const;

int get\_size() const;

void set(Node \*head, Node \*tail, int size);

Node \*remove\_head();

void add(const Element& elmt);

void add(Node \*node);

void clear(); // set head and tail to nullptr and delete all the nodes

void reset(); // set head and tail to nullptr but don't delete the nodes

void print() const;

void split(LinkedList& list1, LinkedList& list2);

void concatenate(LinkedList& other\_list);

private:

Node \*head;

Node \*tail;

int size;

};

#endif /\* LINKEDLIST\_H\_ \*/

ListSorter.cpp

#include <iostream>

#include <vector>

#include "Element.h"

#include "ListSorter.h"

/\*\*

\* Constructor.

\* @param name the name of the algorithm.

\*/

ListSorter::ListSorter(string name) : Sorter(name) {}

/\*\*

\* Destructor.

\*/

ListSorter::~ListSorter() {}

/\*\*

\* Generate the data to sort. The data can be random,

\* already sorted, sorted in reverse, or all zeros.

\* @param generator the data generator.

\* @param size the number of data elements.

\*/

void ListSorter::generate\_data(DataGenerator \*generator, int size)

{

clear\_data();

vector<Element> v;

generator->generate\_data(v, size);

for (int i = 0; i < size; i++) data.add(v[i]);

set\_size(size);

}

/\*\*

\* Print the data elements.

\*/

void ListSorter::print\_data() const

{

Node \*ptr = data.get\_head();

while (ptr != nullptr)

{

cout << " " << ptr->element.get\_value();

ptr = ptr->next;

}

cout << endl;

}

/\*\*

\* Verify that the data is correctly sorted.

\* @return true if correctly sorted, else false.

\*/

bool ListSorter::is\_data\_sorted() const

{

Node \*ptr = data.get\_head();

// Chase next pointers from start to end.

while ((ptr != nullptr) && (ptr->next != nullptr))

{

if (ptr->element.get\_value() > ptr->next->element.get\_value()) return false; // incorrect

ptr = ptr->next;

}

return true; // correct

}

/\*\*

\* Clear the data.

\*/

void ListSorter::clear\_data()

{

if (data.get\_size() > 0) data.clear();

}

ListSorter.h

#ifndef LISTSORTER\_H\_

#define LISTSORTER\_H\_

#include <string>

#include "Sorter.h"

#include "LinkedList.h"

/\*\*

\* Abstract base class of the singly linked list sorting algorithms.

\*/

class ListSorter: public Sorter

{

public:

ListSorter(string name);

virtual ~ListSorter();

virtual void generate\_data(DataGenerator \*generator, int size);

void print\_data() const;

void clear\_data();

protected:

LinkedList data; // the data to sort

bool is\_data\_sorted() const;

};

#endif /\* LISTSORTER\_H\_ \*/

MergeSort.h

#ifndef MERGESORT\_H\_

#define MERGESORT\_H\_

#include <string>

#include "LinkedList.h"

#include "ListSorter.h"

/\*\*

\* The class that implements the mergesort algorithm

\* for a singly linked list of data.

\*/

class MergeSort: public ListSorter

{

public:

MergeSort(string name);

virtual ~MergeSort();

virtual void run\_sort\_algorithm() throw (string);

private:

void mergesort(LinkedList& list);

void merge(LinkedList& list, LinkedList& sublist1, LinkedList& sublist2);

void clear();

};

#endif /\* MERGESORT\_H\_ \*/

QuickSortOptimal.h

#ifndef QUICKSORTOPTIMAL\_H\_

#define QUICKSORTOPTIMAL\_H\_

#include "QuickSorter.h"

/\*\*

\* The class that implements the optimal quicksort algorithm

\* for a vector of data by using a good pivot strategy.

\*/

class QuickSortOptimal: public QuickSorter

{

public:

QuickSortOptimal(string name);

virtual ~QuickSortOptimal();

private:

virtual Element& choose\_pivot\_strategy(const int left, const int right);

};

#endif /\* QUICKSORTOPTIMAL\_H\_ \*/

QuickSortSuboptimal.h

#ifndef QUICKSORTSUBOPTIMAL\_H\_

#define QUICKSORTSUBOPTIMAL\_H\_

#include "QuickSorter.h"

/\*\*

\* The class that implements the suboptimal quicksort algorithm

\* for a vector of data by using a bad pivot strategy.

\*/

class QuickSortSuboptimal: public QuickSorter

{

public:

QuickSortSuboptimal(string name);

virtual ~QuickSortSuboptimal();

private:

virtual Element& choose\_pivot\_strategy(const int left, const int right);

};

#endif /\* QUICKSORTSUBOPTIMAL\_H\_ \*/

QuickSorter.h

#ifndef QUICKSORTER\_H\_

#define QUICKSORTER\_H\_

#include <string>

#include "VectorSorter.h"

#include "Element.h"

/\*\*

\* The abstract base class of the quicksort algorithms.

\*/

class QuickSorter: public VectorSorter

{

public:

QuickSorter(string name);

virtual ~QuickSorter();

virtual void run\_sort\_algorithm() throw (string);

protected:

virtual Element& choose\_pivot\_strategy(const int left, const int right) = 0;

private:

void quicksort(const int left, const int right);

int partition(const int left, const int right, const Element& pivot);

Element& choose\_pivot(const int left, const int right);

};

#endif /\* QUICKSORTER\_H\_ \*/

SelectionSort.cpp

#include "SelectionSort.h"

/\*\*

\* Constructor.

\* @param name the name of this algorithm.

\*/

SelectionSort::SelectionSort(string name) : VectorSorter(name) {}

/\*\*

\* Destructor.

\*/

SelectionSort::~SelectionSort() {}

/\*\*

\* Run the selection sort algorithm.

\* @throws an exception if an error occurred.

\*/

void SelectionSort::run\_sort\_algorithm() throw (string)

{

int sizem1 = size - 1;

// For each element of the vector ...

for (int i = 0; i < sizem1; i++)

{

Element smallest = data[i];

int index\_of\_smallest = i;

// ... find the index of the smallest value

// in the rest of the vector.

for (int j = i + 1; j < size; j++)

{

compare\_count++;

if (data[j] < smallest)

{

smallest = data[j];

index\_of\_smallest = j;

}

}

// Swap in the smallest value if it isn't already there.

if (i != index\_of\_smallest) swap(i, index\_of\_smallest);

}

}

SelectionSort.h

#ifndef SELECTIONSORT\_H\_

#define SELECTIONSORT\_H\_

#include "VectorSorter.h"

/\*\*

\* The class that implements the selection sort algorithm

\* for a vector of data.

\*/

class SelectionSort: public VectorSorter

{

public:

SelectionSort(string name);

virtual ~SelectionSort();

virtual void run\_sort\_algorithm() throw (string);

};

#endif /\* SELECTIONSORT\_H\_ \*/

ShellSortOptimal.h

#ifndef SHELLSORTOPTIMAL\_H\_

#define SHELLSORTOPTIMAL\_H\_

#include "VectorSorter.h"

/\*\*

\* The class that implements the optimal shellsort algorithm

\* for a vector of data, where the diminishing increment is

\* calculated according to Don Knuth.

\*/

class ShellSortOptimal: public VectorSorter

{

public:

ShellSortOptimal(string name);

virtual ~ShellSortOptimal();

virtual void run\_sort\_algorithm() throw (string);

};

#endif /\* SHELLSORTOPTIMAL\_H\_ \*/

ShellSortSuboptimal.h

#ifndef SHELLSORTSUBOPTIMAL\_H\_

#define SHELLSORTSUBOPTIMAL\_H\_

#include "VectorSorter.h"

/\*\*

\* The class that implements the suboptimal Shellsort algorithm

\* for a vector of data, where the diminishing increment is halved

\* for each pass.

\*/

class ShellSortSuboptimal: public VectorSorter

{

public:

ShellSortSuboptimal(string name);

virtual ~ShellSortSuboptimal();

virtual void run\_sort\_algorithm() throw (string);

};

#endif /\* SHELLSORTSUBOPTIMAL\_H\_ \*/

SortTests.cpp

//SAMPLE

#include <iostream>

#include <iomanip>

#include <string>

#include <ctime>

#include <chrono>

#include "DataRandom.h"

#include "DataSorted.h"

#include "DataReverseSorted.h"

#include "DataAllZeros.h"

#include "SelectionSort.h"

#include "InsertionSort.h"

#include "ShellSortSuboptimal.h"

#include "ShellSortOptimal.h"

#include "QuickSortSuboptimal.h"

#include "QuickSortOptimal.h"

#include "MergeSort.h"

using namespace std;

using namespace std::chrono;

bool do\_sort(Sorter \*sorter, int n, DataGenerator \*generator);

void output\_sort\_stats(bool sorted, const Sorter \*sorter);

string commafy(long n);

/\*\*

\* Main.

\*/

int main()

{

bool sorted;

// Number of data items to test.

int N[] = {10, 100, 1000, 10000};

// Sorting algorithms.

Sorter \*sorters[] =

{

new SelectionSort("Selection sort"),

new InsertionSort("Insertion sort"),

new ShellSortSuboptimal("Shellsort suboptimal"),

new ShellSortOptimal("Shellsort optimal"),

new QuickSortSuboptimal("Quicksort suboptimal"),

new QuickSortOptimal("Quicksort optimal"),

new MergeSort("Mergesort"),

};

// Data generators.

DataGenerator \*generators[] =

{

new DataRandom(),

new DataSorted(),

new DataReverseSorted(),

new DataAllZeros(),

};

steady\_clock::time\_point start\_time = steady\_clock::now();

// Loop for each data generator.

for (DataGenerator \*generator : generators)

{

string generator\_name = generator->get\_name();

cout << endl;

for (int i = 0; i < generator\_name.length(); i++) cout << "=";

cout << endl << generator\_name << endl;

for (int i = 0; i < generator\_name.length(); i++) cout << "=";

cout << endl;

// Loop for each data size.

for (int n : N)

{

cout << endl << "N = " << commafy(n) << endl << endl;

cout << setw(25) << "ALGORITHM" << setw(12) << "COPIES"

<< setw(12) << "DESTRUCTS" << setw(12) << "MOVES"

<< setw(12) << "COMPARES" << setw(12) << "MILLISECS"

<< endl;

// Loop for each sorting algorithm.

for (Sorter \*sorter : sorters)

{

cout << setw(25) << sorter->get\_name();

sorted = do\_sort(sorter, n, generator);

output\_sort\_stats(sorted, sorter);

// Clean up after a sort.

sorter->clear\_data();

}

}

}

// Pick up the garbage.

for (Sorter \*sorter : sorters) delete sorter;

for (DataGenerator \*generator : generators) delete generator;

steady\_clock::time\_point end\_time = steady\_clock::now();

long elapsed\_time = duration\_cast<seconds>(end\_time - start\_time).count();

cout << endl << "Done! " << elapsed\_time << " seconds." << endl;

return 0;

}

/\*\*

\* Perform a sort with a given algorithm and data generator.

\* @param sorter the sorting algorithm.

\* @param n the number of data elements to sort.

\* @param generator the data generator.

\*/

bool do\_sort(Sorter \*sorter, int n, DataGenerator \*generator)

{

// Generate data for the algorithm to sort.

sorter->generate\_data(generator, n);

try

{

return sorter->sort(); // do a sort

}

catch (string& message)

{

cout << endl << message << endl;

return false;

}

}

/\*\*

\* Output a sort algorithm's move and compare counts and elapsed time.

\* @param sorted true if correctly sorted, else false.

\* @param sorter the sorting algorithm.

\*/

void output\_sort\_stats(bool sorted, const Sorter \*sorter)

{

if (sorted)

{

cout << setw(12) << commafy(sorter->get\_copy\_count());

cout << setw(12) << commafy(sorter->get\_destructor\_count());

cout << setw(12) << commafy(sorter->get\_move\_count());

cout << setw(12) << commafy(sorter->get\_compare\_count());

cout << setw(12) << commafy(sorter->get\_elapsed\_ms());

cout << endl;

}

else

{

cout << "\*\*\*\*\* Sort error! \*\*\*\*\*" << endl;

}

}

/\*\*

\* Convert a number to a string with commas.

\* @param n the number.

\*/

string commafy(long n)

{

string str = to\_string(n);

int pos = str.length() - 3;

while (pos > 0)

{

str.insert(pos, ",");

pos -= 3;

}

return str;

}

Sorter.cpp

#include "Sorter.h"

#include "Element.h"

using namespace std;

/\*\*

\* Constructor.

\* @param name the name of the sorting algorithm.

\*/

Sorter::Sorter(string name)

: size(0), move\_count(0), compare\_count(0), name(name), elapsed\_time(0) {}

/\*\*

\* Destructor.

\*/

Sorter::~Sorter() {}

string Sorter::get\_name() const { return name; }

/\*\*

\* Get the data size.

\* @return the size.

\*/

int Sorter::get\_size() const { return size; }

/\*\*

\* Set the data size.

\* @param sz the size to set.

\*/

void Sorter::set\_size(int sz) { size = sz; }

/\*\*

\* Get the number of moves during a sort.

\* @return the number of moves.

\*/

long Sorter::get\_move\_count() const { return move\_count; }

/\*\*

\* Get the number of compares during a sort.

\* @return the number of compares.

\*/

long Sorter::get\_compare\_count() const { return compare\_count; }

/\*\*

\* @return the element copy count.

\*/

long Sorter::get\_copy\_count() const { return Element::get\_copy\_count(); }

/\*\*

\* @return the element destructor count.

\*/

long Sorter::get\_destructor\_count() const { return Element::get\_destructor\_count(); }

/\*\*

\* Sort the data by invoking the sorting algorithm.

\* Count the number of data element moves and compares.

\* Calculate the elapsed time in milliseconds.

\* @throws an exception if an error occurred.

\*/

bool Sorter::sort() throw (string)

{

move\_count = 0;

compare\_count = 0;

Element::reset();

start\_timer();

// Sort the data according to the algorithm

// that is defined by the sorting subclasses.

run\_sort\_algorithm();

stop\_timer();

return is\_data\_sorted();

}

/\*\*

\* Start the timer before beginning a sort.

\*/

void Sorter::start\_timer() { timer = steady\_clock::now(); }

/\*\*

\* Stop the timer after the sort finishes.

\*/

void Sorter::stop\_timer()

{

steady\_clock::time\_point now = steady\_clock::now();

elapsed\_time = duration\_cast<milliseconds>(now - timer).count();

}

/\*\*

\* Calculate the elapsed sort time.

\* @return the elapsed time in milliseconds.

\*/

long Sorter::get\_elapsed\_ms() const { return elapsed\_time; }

Sorter.h

#ifndef SORTER\_H\_

#define SORTER\_H\_

#include <string>

#include <vector>

#include <ctime>

#include <chrono>

#include "DataGenerator.h"

using namespace std;

using namespace std::chrono;

/\*\*

\* The abstract base class of the sorting algorithms.

\*/

class Sorter

{

public:

Sorter(string name);

virtual ~Sorter();

// Name of the sorting algorithm.

string get\_name() const;

// Size (number of data elements) to sort.

int get\_size() const;

void set\_size(int sz);

// Get the number of data element moves and compares

// during a sort, and the elapsed time in milliseconds.

long get\_move\_count() const;

long get\_compare\_count() const;

long get\_elapsed\_ms() const;

long get\_constructor\_count() const;

long get\_copy\_count() const;

long get\_destructor\_count() const;

// Abstract functions to generate the data to sort.

virtual void generate\_data(DataGenerator\* generator, int size) = 0;

// Sort the data. Not abstract!

virtual bool sort() throw (string);

virtual void clear\_data() = 0;

virtual void print\_data() const = 0; // useful for debugging

protected:

int size; // number of data elements to sort

long move\_count; // count of data element moves during a sort

long compare\_count; // count of data element compares during a sort

// Abstract functions to run a sort algorithm

// and to verify a correct sort.

virtual void run\_sort\_algorithm() throw (string) = 0;

virtual bool is\_data\_sorted() const = 0;

private:

string name;

steady\_clock::time\_point timer;

long elapsed\_time;

void start\_timer();

void stop\_timer();

};

#endif /\* SORTER\_H\_ \*/

VectorSorter.cpp

#include <iostream>

#include <ctime>

#include "VectorSorter.h"

VectorSorter::VectorSorter(string name) : Sorter(name) {}

/\*\*

\* Destructor.

\*/

VectorSorter::~VectorSorter() {}

/\*\*

\* Generate the data to sort. The data can be random,

\* already sorted, sorted in reverse, or all zeros.

\* @param generator the data generator.

\* @param size the number of data elements.

\*/

void VectorSorter::generate\_data(DataGenerator \*generator, int size)

{

clear\_data();

generator->generate\_data(data, size);

set\_size(size);

}

/\*\*

\* Print the data elements.

\*/

void VectorSorter::print\_data() const

{

int size = get\_size();

for (int i = 0; i < size; i++) cout << " " << data[i];

cout << endl;

}

/\*\*

\* Verify that the data is correctly sorted.

\* @return true if correctly sorted, else false.

\*/

bool VectorSorter::is\_data\_sorted() const

{

int sizem1 = get\_size() - 1;

for (int i = 0; i < sizem1; i++)

{

if (data[i] > data[i+1]) return false; // incorrect

}

return true; // correct

}

/\*\*

\* Clear the data.

\*/

void VectorSorter::clear\_data()

{

if (data.size() > 0) data.clear();

}

/\*\*

\* Exchange the values of two data elements.

\* Counts as two moves.

\*/

void VectorSorter::swap(const int index1, const int index2)

{

Element temp = data[index1];

data[index1] = data[index2];

data[index2] = temp;

move\_count += 2;

}

VectorSorter.h

#ifndef VECTORSORTER\_H\_

#define VECTORSORTER\_H\_

#include <string>

#include <vector>

#include "Sorter.h"

#include "Element.h"

using namespace std;

/\*\*

\* Abstract base class of the vector sorting subclasses.

\*/

class VectorSorter: public Sorter

{

public:

VectorSorter(string name);

virtual ~VectorSorter();

virtual void generate\_data(DataGenerator \*generator, int size);

void clear\_data();

void print\_data() const;

protected:

vector<Element> data; // the vector to sort

bool is\_data\_sorted() const;

void swap(const int index1, const int index2);

};

#endif /\* VECTORSORTER\_H\_ \*/